CS522 - Programming Language Semantics

Maude, Initial Models and Recursive/Inductive Definitions

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

General Information about Maude

Maude is an *executable specification language* supporting *rewrite logic specifications* and having at its core a very fast *rewrite engine*. We will use Maude as a platform for executing programming language definitions following various definitions styles. Our short term objectives are:

1. Learn how to use Maude;
2. Show how various language definitional styles, including the ones discussed so far (big-step and small-step SOS, MSOS and context reduction) can be formally defined in Maude;

Maude's roots go back to Clear (Edinburgh) in 1970s and to OBJ (Stanford, Oxford) in 1980s. There are many other languages in the same family, such as CafeOBJ (Japan), BOBJ (San Diego,

California), ELAN (France).

Maude is currently being developed at Stanford Research Institute and at the University of Illinois at Urbana-Champaign. You can find more about Maude, including a manual and many papers and examples, on its webpage at `http://maude.cs.uiuc.edu`.

We will use Maude to specify programming language features, which, when put together via simple Maude module operations, lead to specifications of programming languages. Due to the fact that Maude is executable, *interpreters for programming languages designed this way will be obtained for free*, which will be very useful for understanding, refining and/or changing the languages.

Moreover, *analysis tools* for the specified languages can also be obtained with little effort, such as type checkers, abstract interpreters or model-checkers, either by using analysis tools

already provided by Maude or by defining them explicitly on top of a language definition.

Devising *formal semantic executable models* of desired languages or tools *before these are implemented* is a crucial step towards a deep understanding of the language or tool.

In simplistic terms, it is like devising a simulator for an expensive system before building the actual system. However, our "simulators" in this class will consist of exactly the *semantics*, or the *meaning*, of our desired systems, defined using a very rigorous, mathematical notation. In the obtained formal executable model of a programming language, "executing a program" will correspond to nothing but *logical inference* within the semantics of the language.

Maude should run on the assigned machines. Type `maude` and you should see a welcome screen like the following:

```
bash$ maude
     \||||||||||||||||||/
   --- Welcome to Maude ---
     /||||||||||||||||||\
    Maude 2.4 built: Dec  9 2008 20:35:33
    Copyright 1997-2008 SRI International
   Wed Sep  2 11:10:08 2009
Maude>
```

It can also be easily installed on almost any platform. Download it from its web page. Also print its user manual. You will need it.

How to Use Maude

Maude is interpreted, so you can just type your specifications and commands. However, a better way is to just type everything in one file, say `p.maude`, and then include that file with the command "`Maude> in p`". Use "`quit`" or simply "`q`", to quit.

You can correct/edit your Maude file and then load it with "`Maude> in p`" as many times as needed. However, keep in mind that Maude maintains only one working session, in particular one module database, until you quit it. This can sometimes lead to "unexpected" errors for beginners, so if you are not sure about an error just quit and then restart Maude.

Modules

Specifications are introduced as *modules*, or *theories*. There are several kinds of modules, but for simplicity we will use only general modules in this class, which have the syntax

```
mod <NAME> is
  <BODY>
endm
```

where `<NAME>` can be any identifier, typically using capital letters.

The `<BODY>` of a module can include importation of other modules, sort and operation declarations, and a set of sentences. The sorts together with the operations form the *signature* of that module, and represent *the interface of that module to the outside world*.

We will use modules to define *programming language features*. These features can be then imported by other modules, which can extend them or use them to define other features. A programming language will be also defined as a module, namely one importing all the modules defining its desired features.

Let us exemplify this paradigm by defining the simplest programming language that we will see in this class. It is so simple, that it does not even deserve to be called a "programming language", but we do so for the sake of a uniform terminology.

The language specified in what follows defines *natural numbers with addition and multiplication*. One module will define the feature "addition" and another will define the feature "multiplication" on top of addition.

An Example: Peano Natural Numbers

The following defines natural numbers with successor and addition, using Peano's axiomatization:

```
mod PEANO-NAT is
  sort Nat .
  op zero : -> Nat .
  op succ : Nat -> Nat .
  op plus : Nat Nat -> Nat .
  vars N M : Nat .
  eq plus(zero, M) = M .
  eq plus(succ(N), M) = succ(plus(N, M)) .
endm
```

Declarations are separated by periods, which should typically have white spaces before and after.

Signatures

One sort, `Nat`, and three operations, `zero`, `succ`, and `plus`, form the signature of `PEANO-NAT`. Sorts are declared with the keywords `sort` or `sorts`, and operations with `op` or `ops`.

The three operations have zero, one and two arguments, resp., whose sorts are between `:` and `->`. Operations of zero arguments are also called *constants*, those of one argument are called *unary* and those of two *binary*. The result sort appears after `->`.

Use `ops` when two or more operations of same arguments are declared together, and use white spaces to separate them:

```
ops plus mult : Nat Nat -> Nat .
```

There are few special characters in Maude. If you use `op` in the above then only one operation, called "`plus mult`", is declared.

Equations

The two equations in `PEANO-NAT` are "properties", or constraints, that these operations should satisfy. More precisely, any *correct* implementation of Peano natural numbers should satisfy them.

All equations are quantified universally, so we should read "for all `M` and `N` of sort `Nat`, `plus(s(N), M) = s(plus(N,M))`". All equations in a module can be used in reasoning about the defined structures.

Exercise 1 *Prove that any correct implementation of* `PEANO-NAT` *should satisfy the property*

```
plus(succ(succ(zero)), succ(succ(succ(zero)))) =
 plus(succ(succ(succ(succ(zero)))), succ(zero)).
```

Equations can be applied from *left-to-right* or from *right-to-left* in reasoning, which means that equational proofs may require exponential search, thus making them theoretically intractable.

Rewriting

Maude regards all equations as *rewriting rules*, which means that they are applied only from left to right. Thus, any well-formed term can either be derived infinitely often, or be reduced to a *normal form*, which cannot be reduced anymore by applying equations as rewriting rules.

Maude's command to reduce a term to its normal form is `reduce` or simply `red`. Reduction will be made in the last defined module:

```
Maude> reduce plus(succ(succ(zero)), succ(succ(succ(zero)))) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: succ(succ(succ(succ(succ(zero)))))
Maude>
```

Millions of rewrites per second can be performed, for which reason Maude can be (almost) used as a programming language.

Importation

Modules can be imported, using the keywords `protecting`, `extending` or `including`. The difference between these importation modes is subtle and semantical rather than operational. Fell free to use `including`, the most general of them, all the time and read the manual for more information.

The following module extends `PEANO-NAT` with multiplication:

```
mod PEANO-NAT* is
  including PEANO-NAT .
  op mult : Nat Nat -> Nat .
  vars M N : Nat .
  eq mult(zero, M) = zero .
  eq mult(succ(N), M) = plus(mult(N, M), M) .
endm
```

Variable declarations are *not* imported.

Putting it all Together

We can now define our simple programming language and "execute programs" (in this case, PEANO-NAT* already had all the features):

```
mod SIMPLEST-PROGRAMMING-LANGUAGE is
  including PEANO-NAT .
  including PEANO-NAT* .
endm
red mult(succ(succ(zero)), mult(succ(succ(zero)), succ(succ(zero)))) .
```

The following is Maude's output:

```
reduce in SIMPLEST-PROGRAMMING-LANGUAGE :
  mult(succ(succ(zero)), mult(succ(succ(zero)), succ(succ(zero)))) .
rewrites: 16 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: succ(succ(succ(succ(succ(succ(succ(succ(zero))))))))
```

Even though this language is *very simple* and its syntax is *ugly*, it

nevertheless shows a formal and executable definition of a language using equational logic and rewriting.

Our languages or analysis tools discussed in this class will be defined in a relatively similar manner, though, as expected, they will be more involved. For example, a notion of state will be needed, as well as several control structures.

The Mix-fix Notation

Some operations are prefix, but others are infix, postfix, or *mix-fix*, i.e., arguments can occur anywhere, like in the case of `if_then_else_`.

Maude supports mix-fix notation by allowing the user to place the arguments of operations wherever one wants by using underscores. Here are some examples:

```
op _+_ : Int Int -> Int .
op _! : Nat -> Nat .
op in_then_else_ : BoolExp Stmt Stmt -> Stmt .
op _?_:_ : BoolExp Exp Exp -> Exp .
```

Exercise 2 *Rewrite* `PEANO-NAT` *and* `PEANO-NAT*` *using mix-fix notation. What happens if you try to reduce an expression containing both* `_+_` *and* `_*_` *without parentheses?*

BNF (Bachus-Naur Form) or CFG (Context Free Grammar) notations to describe the syntax of programming languages are very common. We also used BNF to define the syntax of our simple language in the lectures on SOS. From here on, we can interchangeably use the mix-fix notation instead of BNF or CFG, so it is important to understand the relationship between the mix-fix notation and BNF and CFG:

Exercise 3 *Argue that the mix-fix notation is equivalent to BNF and CFG. Find an interesting simple example language and express its syntax using the mix-fix notation, BNF, and production-based CFG (e.g.,* $A \to AB \mid a$ *and* $B \to BA \mid b$*).*

Parsing

Mix-fix notation leads to an interesting problem, that of *parsing*. In the modified `PEANO-NAT*` of Ex. 2, how do we parse `x + y * z`?

Maude has a command called `parse`, which parses a term using the syntax of the most recently defined module. To see *all* the parentheses, first type the command "`set print with parentheses on .`" (see Subsection 17.5 in Maude's manual).

"`parse x + y * z .`" reports ambiguous parsing. Use parentheses to disambiguate parsing, such as "`parse x + (y * z) .`". To give priorities to certain operators in order to reduce the number of parentheses, use *precedence attribute* when you declare operators:

```
op _+_ : Int Int -> Int [prec 33] .
op _*_ : Int Int -> Int [prec 31] .
```

The lower the precedence the stronger the binding!

Maude has several *builtin modules*, including ones for natural and integer numbers. The operations `_+_` and `_*_` have precedences 33 and 31, respectively, in these builtin modules. With these precedences, one can parse as follows:

```
Maude> set print with parentheses on .
Maude> parse -10 + 2 * 3 .
Int: (-10 + (2 * 3))
```

Sometimes, especially when debugging, the mixfix notation may be confusing. If you wish to turn it off, use the command:

```
Maude> set print mixfix off .
Maude> parse -10 + 2 * 3 .
Int: _+_(-10, _*_(2, 3))
```

Associativity, Commutativity and Identity Attributes

Many of the binary operations that will be used in this class will be associative (A), commutative (C) or have an identity (I), or combinations of these. E.g., `_+_` is associative, commutative and has `0` as identity. All these can be added as attributes to operations when declared:

```
op _+_ : Int Int -> Int [assoc comm id: 0 prec 33] .
op _*_ : Int Int -> Int [assoc comm id: 1 prec 31] .
```

Notice that each of the A, C, and I attributes are logically equivalent to appropriate equations, such as

```
eq A + (B + C) = (A + B) + C .
eq A + B = B + A .  ---> attention: rewriting does not terminate!
eq A + 0 = A .
```

Then why are the ACI attributes necessary? For several reasons:

1. Associativity removes the need for useless parentheses:

```
Maude> parse -10 + 2 + 3 .
Int: -10 + 2 + 3
```

2. Commutativity will allow rewriting to terminate. The normal form will, however, be *modulo commutativity*.

3. ACI matching is perhaps the most interesting reason. It will be described next.

Matching Modulo Attributes

Lists occur in many programming languages. The following module is a an elegant but tricky specification for lists of integers with a membership operation _in_:

```
mod INT-LIST is protecting INT .
  sort IntList .
  subsort Int < IntList .
  op nil : -> IntList .
  op __ : IntList IntList -> IntList [assoc id: nil] .
  op _in_ : Int IntList -> Bool .
  var I : Int .  vars L L' : IntList .
  eq I in L I L' = true .
  eq I in L = false [owise] .
endm
```

Note the declaration "`subsort Int < IntList .`", which says that integers are also lists of integers. Then the constant `nil` and the concatenation operation `__` can generate any finite list of integers:

```
Maude> parse 1 2 3 4 5 .
IntList: 1 2 3 4 5
Maude> red 1 nil 2 nil 3 nil 4 nil 5 6 7 nil .
result IntList: 1 2 3 4 5 6 7
```

Further, by AI matching, the membership operation can be defined without having to traverse the list element by element!

```
Maude> red 3 in 2 3 4 .
result Bool: true
Maude> red 3 in 3 4 5 .
result Bool: true
Maude> red 3 in 1 2 4 .
result Bool: false
```

By AI matching, `I`, `L`, and `L'` in the left-hand-side term of "`eq I`

`in L I L' = true .`" can match any integer or lists of integers, respectively, including `nil`. Since `I` can only be matched by `3`, the other matches follow automatically.

The attribute `owise` in "`eq I in L = false [owise] .`" tells Maude to apply that equation only "otherwise", that is, only if no other equation, in this case the previous one only, can be applied.

If one wants to define sets of integers, then, besides replacing the sort `IntList` probably by `IntSet`, one has to declare the concatenation also commutative, and one has to replace the first equation by "`eq I in I L = true .`".

Exercise 4 *Define a Maude module called* `INT-SET` *specifying sets of integers with membership, union, intersection and difference (elements in one set and not in the other).*

The matching modulo attributes are implemented very efficiently in Maude, so you (typically) don't have to worry about efficiency.

Built-in Modules

There are several built-in modules. We will use the following:

- `BOOL`. Provides a sort `Bool` with two constants `true`, `false` and basic boolean calculus, together with `if_then_else_fi` and `_==_` operations. The later takes two terms, reduces them to their normal forms, and then returns `true` iff they are equal (modulo ACI, as appropriate); otherwise it returns `false`.
- `INT`. Provides a sort `Int`, arbitrary large integers as constants of sort `Int`, together with the usual operations on these.
- `QID`. Provides a sort `Qid` together with arbitrary large quoted identifiers, as constants of sort `Qid`, of the form: `'a`, `'b`, `'a-larger-identifier`, etc.

To see an existing module, built-in or not, type the command

```
Maude> show module <NAME> .
```

For example, "`show module BOOL .`" will output

```
mod BOOL is protecting TRUTH .
  op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
  op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
  op _xor_ : Bool Bool -> Bool [assoc comm prec 57] .
  op not_ : Bool -> Bool [prec 53] .
  op _implies_ : Bool Bool -> Bool [prec 61 gather (e E)] .
  vars A B C : Bool .
  eq true and A = A .  eq false and A = false .
  eq A and A = A .     eq false xor A = A .
  eq A xor A = false . eq not A = A xor true .
  eq A and (B xor C) = A and B xor A and C .
  eq A or B = A and B xor A xor B .
  eq A implies B = not (A xor A and B) .
endm
```

Exercise 5 *Use the command "`show module <NAME> .`" on the three modules above and try to understand them.*

On Models

From now on we may use the word *model* to refer to implementations. This terminology comes from mathematical logics and "model theory", so it has a more mathematical flavor. Anyhow, we should think of both implementations and models as "realizations" of specifications.

A specification can have therefore several models: all those satisfying its properties. However, not all models are always intended. In this lecture we discuss *initial models*, for which *induction* is a valid proof technique, and then we discuss the relationship between these and *recursively defined operations*.

Bad Models

Among the models of a specification, there are some which have quite unexpected properties. For example, a simple specification of lists of bits without any equations, can be defined as follows:

```
mod BIT-LIST is
  sorts Bit BitList .  subsort Bit < BitList .
  ops 0 1 : -> Bit .
  op nil : -> BitList .
  op _,_ : Bit BitList -> BitList .
endm
```

One possible model, $\mathcal{M}$, can do the following:

- Implement elements of sort `BitList` as *real numbers* (yes, it looks strange but there is nothing to forbid this so far),
- Implement `nil` and `0` as *0*, and `1` as *1*,
- Implement `_,_` as *addition*.

The model $\mathcal{M}$ has very strange properties, such as

Junk

There are lots of "lists" in $\mathcal{M}$ which are not intended to be lists of bits, such as, for example, the number π.

Confusion

There are distinct lists which in fact collapse when interpreted in $\mathcal{M}$, such as, for example, the lists `0,1` and `1,0`. Concatenation becomes commutative in $\mathcal{M}$, which is highly undesirable.

Initial Models

Initial models are those with *no junk* and *no confusion*. How can we define such a model?

For the specification of lists of bits above, we can build an initial model $\mathcal{T} = (\mathcal{T}_{\texttt{Bit}}, \mathcal{T}_{\texttt{BitList}})$ as the pair of *smallest* sets with the following properties:

1. `0` and `1` belong to $\mathcal{T}_{\texttt{Bit}}$;
2. $\mathcal{T}_{\texttt{Bit}}$ is included in $\mathcal{T}_{\texttt{BitList}}$;
3. `nil` is an element of $\mathcal{T}_{\texttt{BitList}}$, and `0,L` and `1,L` are elements of $\mathcal{T}_{\texttt{BitList}}$ whenever `L` is an element of $\mathcal{T}_{\texttt{BitList}}$.

The model $\mathcal{T}$ is exactly the desired model for lists of bits: contains nothing but lists and no two distinct lists are collapsed!

One can similarly define an initial model for any signature. Intuitively, initial models of signatures consist of *exactly* all the well-formed terms over the syntax specified by the signature.

In the case of specifications, which contain not only signatures but also sentences, initial models can be defined as well. Essentially, they are defined by

> *taking the initial models of the corresponding signatures and collapsing all terms which can be shown equal using specification's sentences*.

As an example, let us reconsider the specification of natural numbers `PEANO-NAT`:

```
mod PEANO-NAT is
  sort Nat .
  op zero : -> Nat .
  op succ : Nat -> Nat .
  op plus : Nat Nat -> Nat .
  vars N M : Nat .
  eq plus(zero, M) = M .
  eq plus(succ(N), M) = succ(plus(N, M)) .
endm
```

The initial model of its signature contains all the well-formed terms built over `zero`, `succ` and `plus` (so no variables!). When we also consider the two equations, many distinct such terms will collapse, because they become equal modulo those equations.

E.g., `plus(succ(succ(zero)), succ(succ(succ(zero))))` is equal to `plus(succ(succ(succ(succ(zero)))), succ(zero))`, so they represent only one element in the initial model of `PEANO-NAT`.

The set of all terms which are equivalent modulo the equations of a specification are typically called *equivalence classes*. Therefore, initial models of specifications have equivalence classes as elements.

From now on in this class, when we talk about models or implementations of specifications, we will actually mean *initial models*. Also, when we prove properties of specifications, we prove them as if for their initial models.

Induction

Why are we interested in initial models? Because in these models, *induction*, a very powerful proof technique very related to recursive definitions that will be intensively used in defining programming language features later in the course, is a valid proof technique.

Where is the Mistake?

To understand the important concept of initial model as well as its relationship to induction better, let us play a "where is the mistake" game. We prove by induction a property of a specification, and then we show that there are implementations which satisfy the specification but which do *not* satisfy the "proved" property.

We first prove that commutativity of _+_ is a consequence of the specification of Peano natural numbers, this time using the mix-fix notation:

```
mod PEANO-NAT is sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
endm
```

To show that `M + N = N + M` for all natural numbers `M` and `N`, we can do a proof by *induction* on either `M` or `N`, say `M`. Since any natural number is either `0` or the successor of a smaller natural number, we need to analyze two cases:

Case 1: `M = 0`. We need to prove that `0 + N = N + 0` for any natural number `N`. By the first equation, we only need to show that for any natural number `N`, it is the case that `N + 0 = N`. The only way to show it is by induction again, this time by `N`. There are two cases again:

Case 1.1: `N = 0`. We have to show that `0 + 0 = 0`, which follows by the first equation.

Case 1.2: Assume `n + 0 = n` for some `n` and prove that `s(n) + 0 = s(n)`. By the second equation, `s(n) + 0 = s(n + 0)`, and by the induction's hypothesis, `s(n + 0) = s(n)`. Done.

Case 2: Assume that `m + N = N + m` for some `m` and for any `N`, and prove that `s(m) + N = N + s(m)` for any `N`. By the second equation and the induction hypothesis, it follows that all what is left to prove is `N + s(m) = s(N + m)`, which we prove again by induction:

Case 2.1: `N = 0`. The equality `0 + s(m) = s(0 + m)` is immediate because of the first equation.

Case 2.2: Assume that `n + s(m) = s(n + m)` for some `n` and show `s(n) + s(m) = s(s(n) + m)`. Indeed, `s(n) + s(m) = s(n + s(m)) = s(s(n + m))` by the second equation and the induction hypothesis, and `s(s(n) + m) = s(s(n + m))` by the second equation.

Therefore, by several applications of induction we proved that addition on Peano natural numbers is commutative. Hence, one would naturally expect that addition should be commutative for *any* implementation of natural numbers satisfying the Peano axioms in `PEANO-NAT`. Well, let us consider the following implementation $\mathcal{S}$:

- Natural numbers are interpreted as strings;
- `0` is interpreted as the empty string;
- `s(N)` is the string `aN`, that is, the character `a` concatenated with the string `N`;
- `_+_` is implemented as string concatenation.

$\mathcal{S}$ obviously satisfies the two axioms of `PEANO-NAT`. However, addition is *not* commutative in $\mathcal{S}$! Where is the problem? What's going on here? There is nothing wrong, just that

> *Proofs by induction are NOT valid for all possible models/implementations, but only for special ones!*

At this stage, one can admittedly say: well, but in the string model you can only build "naturals" by adding an 'a' in front of a string, so you can only get strings of a's, in which case addition is commutative. That means that one actually had in mind a special

model all the time, namely one in which all the numbers need to be "reached" starting with zero and applying successor operations. For this model, induction is indeed a valid proof principle, because this model is the initial one. But note that there is no axiom in the Peano definition of natural numbers saying that in any model natural numbers can only be obtained starting with a zero and applying a finite number of successor operations. That was only implicit in our mind, but not explicit in the axioms.

One thing to keep in mind when we talk about specifications is that they give us a set of *constraints* that the possible acceptable worlds should satisfy. They *do not place us in a unique world*. Also, in the context of programming language design, we do not want to restrict too much the class of implementations by design! One implementation of a programming language can be by using a compiler, another by using an interpreter, yet another by translating it into a different programming language.

When one proves a property about a specification, that property is expected to hold in *all* possible worlds. However, our major points above are:

1. Properties one proves by induction are *not* valid in all possible worlds, so one has to restrict the possible universes to those in which induction is valid! Those are called "reachable" in the literature, but this goes beyond the scope of the class. What's important is that the initial model is reachable; that's the model you should have in mind when you write executable language specifications.
2. Once one accepts the restricted set of possible universes in which induction is valid, then one can also have a methodological clean way to develop specifications: think of what are the constructors that you would use in doing proofs by induction, and then make sure you define your new operations for *each of those*.

From here on, we will consider only specifications whose intended models are initial, so proofs by induction are valid.

Exercise 6 *Show that addition is associative in* `PEANO-NAT` *and that multiplication is commutative and associative in* `PEANO-NAT*`*, where we also replace* `mult` *by its mix-fix variant* `_*_`*. These proofs need to be done by induction. Describe also a model/implementation of* `PEANO-NAT*`*, where multiplication is implemented in such a way that it is neither commutative nor associative. You can extend the one on strings if you wish.*

Constructor versus Defined Operations

When we defined the operation `plus` on Peano natural numbers, we defined it recursively on numbers of the form `zero` and `succ(N)`. Why did we do that, considering that `PEANO-NAT` had declared three operations:

```
op zero : -> Nat .
op succ : Nat -> Nat .
op plus : Nat Nat -> Nat .
```

Intuitively, the reason is that because we want `plus` to be completely *defined* in terms of `zero` and `succ`. Formally, this means that any term over the syntax `zero`, `succ`, and `plus` can be shown, using the given equations, equal to a term containing only `zero` and `succ` operations, that is, `zero` and `succ` alone are sufficient to build any Peano number. For that reason, they are called *constructors*.

Defining Operations using Constructors

There is no silver-bullet recipe on how to define "defined" operators, but essentially the main (safe) idea is to

Define the operator's "behavior" on each constructor.

That is what we did when we defined `plus` in `PEANO-NAT` and `mult` in `PEANO-NAT*`: we first defined them on `zero` and then on `succ`.

In general, if `c1`, ..., `cn` are the intended constructors of a data-type, in order to define a new operation `d`, make sure that all equations

```
eq d(c1(X1,...)) = ...
...
eq d(cn(Xn,...)) = ...
```

are in the specification. This gives no guarantee (e.g., one can "define" `plus` as `plus(succ(M),N) = plus(succ(M),N)`), but it is a good enough principle to follow.

Defining Operations on Lists

Let us consider the following specification of lists:

```
mod INT-LIST is protecting INT .
  sort IntList .  subsort Int < IntList .
  op __ : Int IntList -> IntList [id: nil] .
  op nil : -> IntList .
endm
```

Therefore, there are *two constructors for lists*: the empty list and the concatenation of an integer to a list. Let us next define several other important and useful operations on lists. Notice that the definition of each operator treats each of the constructors separately.

The following defines the usual length operator:

```
mod LENGTH is protecting INT-LIST .
  op length : IntList -> Nat .
  var I : Int .  var L : IntList .
  eq length(I L) = 1 + length(L) .
  eq length(nil) = 0 .
endm

red length(1 2 3 4 5) .   ***> should be 5
```

The following defines membership, without speculating matching (in fact, this would not be possible anyway because concatenation is not defined associative as before):

```
mod IN is protecting INT-LIST .
  op _in_ : Int IntList -> Bool .
  vars I J : Int .  var L : IntList .
  eq I in J L = if I == J then true else I in L fi .
  eq I in nil = false .
endm

red 3 in 2 3 4 .  ***> should be true
red 3 in 3 4 5 .  ***> should be true
red 3 in 1 2 3 .  ***> should be true
red 3 in 1 2 4 .  ***> should be false
```

The next operator appends two lists of integers:

```
mod APPEND is protecting INT-LIST .
  op append : IntList IntList -> IntList .
  var I : Int .  vars L1 L2 : IntList .
  eq append(I L1, L2) = I append(L1, L2) .
  eq append(nil, L2) = L2 .
endm

red append(1 2 3 4, 5 6 7 8) .
***> should be 1 2 3 4 5 6 7 8
```

The following imports APPEND and defines an operation which reverses a list:

```
mod REV is protecting APPEND .
  op rev : IntList -> IntList .
  var I : Int .  var L : IntList .
  eq rev(I L) = append(rev(L), I) .
  eq rev(nil) = nil .
endm

red rev(1 2 3 4 5) .   ***> should be 5 4 3 2 1
```

The next module defines an operation which sorts a list of integers by insertion sort:

```
mod ISORT is protecting INT-LIST .
  op isort : IntList -> IntList .
  vars I J : Int .  var L : IntList .
  eq isort(I L) = insert(I, isort(L)) .
  eq isort(nil) = nil .
  op insert : Int IntList -> IntList .
  eq insert(I, J L) = if I > J then J insert(I,L) else I J L fi .
  eq insert(I, nil) = I .
endm

red isort(4 7 8 1 4 6 9 4 2 8 3 2 7 9) .
***> should be 1 2 2 3 4 4 4 6 7 7 8 8 9 9
```

Defining Operations on Binary Trees

Let us now consider a specification of binary trees, where a tree can be either empty or an integer with a left and a right subtree:

```
mod TREE is protecting INT .
  sort Tree .
  op ___ : Tree Int Tree -> Tree .
  op empty : -> Tree .
endm
```

We next define some operations on such trees, also following the structure of the trees given by the two constructors above.

The next simple operation simply mirrors a tree, that is, it recursively replaces each left subtree by the mirrored right subtree and vice-versa:

```
mod MIRROR is protecting TREE .
  op mirror : Tree -> Tree .
  vars L R : Tree .  var I : Int .
  eq mirror(L I R) = mirror(R) I mirror(L) .
  eq mirror(empty) = empty .
endm

red mirror((empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty))
***> should be (empty 2 (empty 6 empty)) 5 ((empty 1 empty) 3 emp
```

Searching in binary trees can be defined as follows:

```
mod SEARCH is protecting TREE .
  op search : Int Tree -> Bool .
  vars I J : Int .  vars L R : Tree .
  eq search(I, L I R) = true .
  eq search(I, L J R) = search(I, L) or search(I, R) [owise] .
  eq search(I, empty) = false .
endm

red search(6, (empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty)) .
***> should be true
red search(7, (empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty)) .
***> should be false
```

Notice the use of the attribute `[owise]` for the second equation.

Exercise 7 *Define* `search` *with only two equations, by using the built-in* `if_then_else_fi`.

Putting Together Trees and Lists

We next define a module which imports both modules of trees and of lists on integers, and defines an operation which takes a tree and returns the list of all integers in that tree, in an infix traversal:

```
mod FLATTEN is
  protecting APPEND .
  protecting TREE .
  op flatten : Tree -> IntList .
  vars L R : Tree .  var I : Int .
  eq flatten(L I R) = append(flatten(L), I flatten(R)) .
  eq flatten(empty) = nil .
endm
red flatten((empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty)) .
***> should be 3 1 5 6 2
```

Exercise 8 *Do the same for prefix and for postfix traversals.*

Exercise 9 *Write an executable specification in Maude that uses binary trees to sort lists of integers. You should define an operation* `btsort : IntList -> IntList`*, which sorts the argument list of integers, as* `isort` *did above. In order to define it, define another operation* `bt-insert : IntList Tree -> Tree`*, which inserts each integer in the list at its place in the tree, and also use the already defined* `flatten` *operation.*